Quiz 1 Review

Today
- Quiz review

Big Picture
Let’s take a moment to review how some of the pieces we’ve looked at so far fit together into recipes for tackling different kinds of software construction problems.

<table>
<thead>
<tr>
<th></th>
<th>methods</th>
<th>mutable objects</th>
<th>ADTs</th>
</tr>
</thead>
<tbody>
<tr>
<td>design</td>
<td>specs</td>
<td>state machine diagram</td>
<td>mutability &amp; immutability</td>
</tr>
<tr>
<td></td>
<td>spec strength</td>
<td>method specs</td>
<td>operations (C,P,O,M)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>datatype definitions</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>functions over datatypes</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>abstraction function &amp; rep invariant</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>equality in ADTs</td>
</tr>
<tr>
<td>test</td>
<td>unit testing</td>
<td>state machine coverage</td>
<td>contextual equivalence</td>
</tr>
<tr>
<td></td>
<td>black box testing</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>input space partitioning</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>boundary values</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>glass box testing</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>code coverage</td>
<td></td>
<td></td>
</tr>
<tr>
<td>code (design patterns)</td>
<td>static checking</td>
<td>enum</td>
<td>abstract syntax tree</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>interpreter pattern</td>
</tr>
<tr>
<td></td>
<td>arrays, lists, maps</td>
<td></td>
<td>checkRep &amp; assertions</td>
</tr>
<tr>
<td></td>
<td>immutable strings</td>
<td></td>
<td>composition &amp; delegation</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>examples</td>
<td>Hailstone</td>
<td>StringBuilder</td>
<td>RatNum, String, List</td>
</tr>
<tr>
<td></td>
<td>pi generation (PS1)</td>
<td>Stack</td>
<td>ImList, Formula</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Markdown parser</td>
<td>Expression (PS2)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Expression parser (PS2)</td>
<td></td>
</tr>
</tbody>
</table>

These lecture notes have been collaboratively authored, with contributions from Saman Amarasinghe, Adam Chlipala, Srini Devadas, Michael Ernst, Max Goldman, John Guttag, Daniel Jackson, Rob Miller, Martin Rinard, and Armando Solar-Lezama. Copyright held by the authors, all rights reserved.
Static Checking

A type is a set of values and operations on those values.

**Static type checking** guarantees that operations are applied only types for which they are defined. The compiler provides this guarantee, statically, before you even run your program.

```
true + true  type error! + is not defined for bool x bool
String s = "";
    s.gotcha();  type error! gotcha method is not defined on String
int x = "5" type error! x is declared as an int, so it can’t point to a String, because that
    might produce a type error somewhere down the line when x is used with
    an int operation
```

Also part of static type checking is ensuring that a class implements an interface provides a method body for every method of the interface.

```
interface ImList<E> {  E first(); ImList<E> rest(); }
class EmptyImList<E> implements ImList<E> { ... }
```

Actual type vs. declared type: variables and expressions have a **declared** type. At runtime, the variables point to an object which has an **actual** type. Those types don’t have to be identical, but they have to be “assignment compatible.” Every method on the declared type has to be found on the actual type.

```
List<Integer>
```

Testing

Testing against the spec

Blackbox vs. glassbox

- Picking input values to powerMod that might cause intermediate steps to overflow int is a glassbox test…
- but it still has to obey the spec! Tests can’t break the contract.

Oracles

- Known answer (pi digits from the web)
- Slow brute-force method (powerMod)
- Comparison that ignores irrelevant details (e.g. order of items, accuracy threshold for output)

Unit testing vs. subsystem/integration/system testing

Input space partitioning

Boundary value testing

```
reverse: String -> String
```

Coverage: all-statements, all-branches, all-paths
Specs

Methods have a contract that separates the caller from the method body

- **precondition**: set of legal arguments and states of the world in which method can be called
- **postcondition**: set of legal return values/exceptions/post-states, given the arguments

(More generally, a class has a contract that separates its users from the internal representation & method behavior)

**Precondition** is expressed as requires clauses. Preconditions are less friendly to clients, because method can do ANYTHING when the precondition isn’t satisfied. e.g.

```java
char getKeypress();
    // requires: user must press a letter key
```

There’s no way for a caller to guarantee this! And if they don’t, what’s the state of the program? Destroyed? Exception thrown? Returns the letter “A”? Returns -1? What?

Some preconditions are unavoidable:

```java
int binarySearch(int x, int[] a)
    // requires: a is sorted in increasing order
```

**Postcondition** can include exceptions – which are essentially like adding to the set of values of the return type. e.g.:

```java
String readLine() throws IOException
```

is similar to returning the union type (using our datatype definition syntax) String + IOException. But note that the exceptional return value also has special control behavior – it can bubble up through levels of method calls until somebody handles it. Regular return values don’t behave that way. If they’re not captured right at the call, they’re ignored.

Checked vs. unchecked exceptions

**Mutators** have a frame condition that specifies what was modified. Think about mutable state in the rep, and think about mutable objects being passed in.

```java
int countElements(Iterator iter);
    // modifies: iter
    // returns: number of elements that were remaining in iter
```

Risks of aliasing

**Immutability**

immutable objects and immutable references
ADTs

An ADT is a datatype defined by its operations
The values of an ADT represent abstract mathematical or physical-world objects

Abstraction function
how the rep fields are mapped to an abstract value
- RatNum
- Complex: polar representation (r, theta) vs. real/imaginary representation
- Card: Suit/int rep vs. Suit/Rank rep vs. int rep (0-51)

Rep invariant – what must always be true of the rep
note that the declared types of the rep already express an invariant which is checked statically
so we’re documenting additional assumptions, and coding them in a runtime check
usually != null is a critical part of every RI
but only for object types, not primitive types

Recursive Data Types

Datatype definition shows how an abstract data type (on the left) is expressed in terms of a rep (on the right)

Simple (non recursive) ADT:
  Stack<E> = List<E>

Relate to A and R
  Stack<E> = E[] x int

Recursive ADT has variants, which are written like constructors:
  Formula = Var(s:String) + Not(f:Formula)

We write an abstract value of a recursive ADT as an expression of constructors
  Not(Not("a"))

and functions can be defined over a recursive data type
  simplify: Formula -> Formula
  simplify(Var(s)) = Var(s)
  simplify(Not(Not(s))) = x
  simplify(Not(Var(s))) = Not(Var(s))

Interpreter pattern implements a function as a method on the interface
- Statically checked, unlike instanceof
Regexes & Grammars

Lexer – takes a sequence of characters and produces a sequence of tokens instead
a token is a unit more useful for parsing, e.g. number, word, identifier, operator
lexer often removes whitespace

Parser – takes a sequence of tokens and produces an abstract syntax tree
an abstract syntax tree is a datatype (usually recursive) representing a phrase in the language

A grammar describes legal phrases in the language using a set of productions (rules)

A regular expression is a single production

Equality

Equivalence relations: reflexive, symmetric, transitive
Abstraction function meaning of equals()
Immutable types must override equals()
ex. define equals() for Formula

Mutable types should not
ex. define equals() for Stack<E>

Relationship between equals() and hashCode()
How to break a HashSet using mutable objects that implement the wrong equals()

Inheritance

Use composition, not inheritance
Substitution principle